Experiment 12
System-on-Chip (SoC) Security Policies

The experiment demonstrates how to implement system-on-chip security policies that protect critical assets from malicious access.

Instructor: Dr. Swarup Bhunia
TAs: Shuo Yang and Reiner Dizon

Theory Background

Security is a critical component of computing systems. With smart computing devices being employed for vast number of highly personalized activities (e.g., shopping, banking, fitness tracking, providing driving directions, etc.), these devices have access to a large amount of sensitive, personal information which must be protected from unauthorized or malicious access. In addition to personalized end-user information, most modern computing systems contain highly confidential collateral from architecture, design, and manufacturing, such as cryptographic and digital rights management (DRM) keys, programmable fuses, on-chip debug instrumentation, defeature bits, etc. Consequently, security architecture, i.e., mechanism to ensure protection of sensitive assets from malicious, unauthorized access, constitutes a crucial component of modern System-on-Chip (SoC) designs.

A significant component of SoC security is driven by the requirement to protect various on-chip assets against unauthorized access. Protection requirements to these security assets can be defined by confidentiality, integrity, and availability properties [1, 2]. The goal of a security policy is to map the requirements to "actionable" design constraints that can be used by SoC designers to develop protection mechanisms. Following are two representative examples of common security policy.

• Example 1: Any IP core in the system-on-chip can only read and write to and from their corresponding memory ranges of the system memory.

• Example 2: In the active crypto mode, the crypto core output cipher text interfaces via the SPI controller are disabled.

Example 1 is an integrity requirement while Example 2 is a confidentiality constraint; The policies provide definitions of (computable) conditions to be satisfied by the design for accessing a security asset. In addition to access control, security policies can capture requirements from information flow, liveness, time-of-check vs. time-of-use (TOCTOU), etc. A primary goal of SoC security architecture is to correctly and efficiently implement the security policies.

Unfortunately, security policies in a modern SoC design are significantly complex and developed in ad hoc manner. A summary of some policy classes is given below.

- **Access Control**: This is the most common class of policies and specifies how different agents in an SoC can access an asset at different points of the execution. Here an "agent" can be a hardware or software component in any IP. Examples 1 and 2 above are examples of such policy. Furthermore, access control forms the basis of many other policies, including information flow, integrity, and secure boot.
- **Information Flow**: Values of secure assets can sometimes be inferred without direct access, through indirect observation or "snooping" of intermediate computation or communications of IPs. Information flow policies restrict such indirect inference. Following is an example:

• **Key Obliviousness**: A low-security IP cannot infer cryptographic keys by snooping only the data from crypto engine on a low-security NoC.

Information flow policies are difficult to analyze. They often require highly sophisticated protection mechanisms and advanced mathematical arguments for correctness, typically involving hardness or complexity results from information security. Consequently, they are employed only on critical assets with very high confidentiality requirements.

- **Liveness**: These policies ensure that the system performs its functionality without "stagnation" throughout its execution. A typical liveness policy is that a request for a resource by an IP is followed by an event response or grant. Deviation from such a policy can result in system deadlock or livelock, consequently compromising system availability requirements.
- **Time-of-Check vs. Time of Use (TOCTOU)**: This refers to the requirement that any agent accessing a resource requiring authorization is indeed the agent that has been authorized. A critical example of TOCTOU is in firmware update, where the policy requires that firmware eventually installed on an update is the same one that has been authenticated.

Case Study

The experiment demonstrates how to implement system-on-chip security policies that protect critical assets from malicious access.

Ensuring authorized access to SoC security assets

To provide better insights about the criticality and significance of secure information and authorized access to assets, an illustrative example is presented in this section with a model SoC. The SoC model is designed with a processor core, a crypto IP, a memory IP, and SPI, a peripheral IP for external communication. The SoC is designed to operate in a manner that only trusted IPs are given access to the secure memory address of memory i.e. an on-chip ram. The crypto IP reads the plaintext and encrypts the text using stored keys. Once the encryption is completed, it stores the data to the trusted memory region that can be accessed by other trusted IPs. Untrusted IPs like the SPI IP is prevented from accessing the memory where secure data is stored.

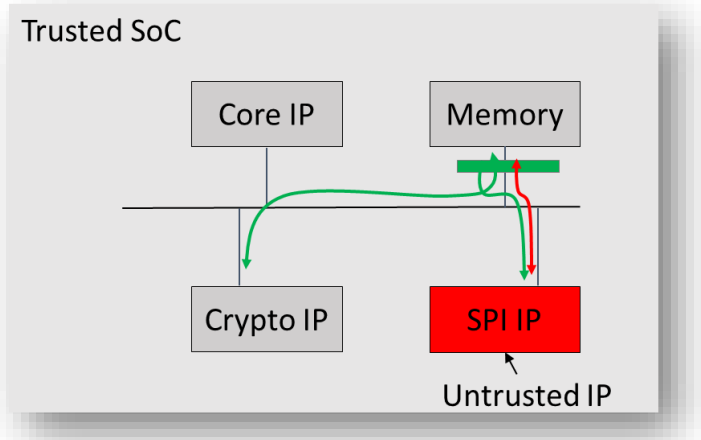

Figure 1. Sample SoC model with trusted and untrusted IP blocks.

Such access control is obtained by implementing security policies in wrapper built around the on-chip memory. The security policy on the memory restricts the access of untrusted IPs like the SPI IP to prevent the leakage of encrypted data. The security policy determines the access privilege of the SPI IP through the address select and lock signal. Figure 2. depicts an example of illustrative micro-architecture of IP ID register. The security mechanism is modeled as a security register with lock bit. The registers of the SoC model are basically flip-flops triggered on the positive edge of the clock. The ID register ``Sec'' only use the lower 5 bits of Data-in and bit 0 of Data-in is used for the lock mechanism. Data-out always reads 8 bits from Read or Lock.

As it works as a secure gateway to access the trusted memory regions, it is crucial to perform thorough security analysis of the registers and fully comprehend the vulnerabilities that might arise from poor design constraints. The ID of the IPs are generated using the 5 lower significant bits of a 8 bit register named ``Sec''. To prevent unauthorized writes on the ``Sec'' register another register named ``Lock'' is added to the design. The LSB of ``Lock'' is used to enable or disable write operations on ``Sec'' register.

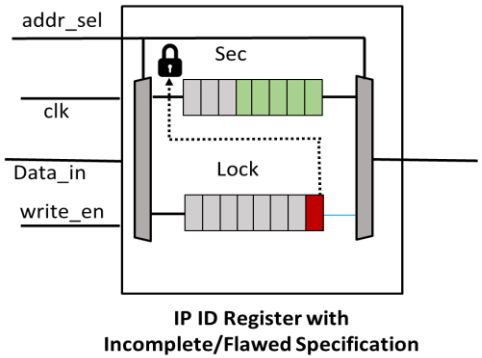

Figure 2. Illustrative example of IP ID register.

A threat model for the illustrative SoC model described above would have several crucial aspects that needs to be considered. For instance, the objective of the threat model is prevention of illegal write operations on the ``Sec'' register when the lock bit of the ``Lock'' register is set to 1. This is a security policy that ensures data integrity property on the ``Sec'' register. The asset for this threat model is the ID and the ``Sec'' register. An attack scenario for the given threat model would be any attempt made by untrusted software to modify the ``Sec'' register once of the lock bit is set. Other properties like confidentiality and availability of ``Sec'' and ``Lock'' registers are trivial for this case study. Once a well-defined threat model is structured, the next task of security analysis is the identification of vulnerabilities. For instance, a closer look at the code snippet shown below will reveal that it is possible to circumvent the data integrity of ``Sec'' register by exploiting poorly written specifications for accessing the ``Sec'' register. Consequently, the bad design and incomplete specification can aid the attacker to modify the lock bit to disable the locking mechanism (shown in figure 2).

On the other hand, a complete set of specification for the design under consideration would protect the write operation on the ``Sec'' register by the ``Lock''. Also, it should be specified that the ``Lock'' register is self-locking. Apart from complete specifications, another critical aspect of designing security features is the definition of right size mitigations. For instance, the security analyst must be able to answer if it is required to design security mechanism for all the registers of the USB IP. If the security features are not adequate, then there's a possibility that some registers of the untrusted IP might contain malicious software. On the other hand, over-protection of the assets can be detrimental to the functional flow of the SoC and can lead to the obsoleteness of security mechanisms.

```
If
        Add_Sel == 0 AND Lock == 1
                Write_En_in = 0
Else
                Write_En_in = Write_En
```

Figure 3. RTL Code with incomplete specification

Experiment Set-up: Configuration

1. The instruments needed for this experiment are the HAHA Board, a USB A to B cable, a USB Blaster, and a computer.
2. The software needed is Quartus, version 15 or higher.
3. Refer to the HAHA User Manual to see the steps of configuring the Altera MAX 10 FPGA.

Experiment Set-up: Instructions

In this experiment, you will need to implement a given security policy into the Altera MAX 10 FPGA, write a similar policy, find the flaws of the policies and write new security policies with correct and complete specifications.

Part I Implement the sample security policy

A sample SoC model is provided in the form of Verilog files on Canvas. The top module is ***Top*** and it can be found in the file ***Top.v***. The sample SoC model has a DLX uP core, a 128-bit AES core, an on-chip ram (128 bit wide and 32 words deep), and a serial peripheral interface (SPI) core. Once the compilation is done, map CLKF, MASRST, lock, and add_sel_aes (AES address select line) on the clock, reset, and two of the external switches of HaHa board respectively. Re-compile the design once the pins are assigned. Then map the design on the FPGA using the device programmer.

Part II Write a similar security policy

Using the lock mechanism (lock and address select line as external switches) write a policy for the controlling the access of SPI to the RAM. For simplicity, use the value assigned to the misoS wire and write the value to the RAM. Then, demonstrate (via In-System Memory Content Editor) that the SPI access to the RAM can be controlled by the lock & address select switch.

Part III Threat Analysis (Identifying the Vulnerability)

Analyze the vulnerability of the sample policies. Try and figure out the security loop hole of the policies and write a new and improved version of the policies that protects the lock register and the security assets of RAM. Refer to the detailed discussion provided in the case study for better comprehension of writing flawless security policies.

Measurement, Calculation, and Question

Answer the following questions.

Part I Implement the sample policy

1) Show the content of the RAM through In-System Memory Content Editor. Take screenshots for the lock switch turned on and off and submit with report.
2) What is a wrapper for hardware IP cores? How does wrappers help in policy implementation through IP reuse?
3) How many logic elements are used to map the sample SoC?

Part II Write a similar security policy

4) Take screenshots of the In-System Memory Content Editor to show the SPI access control policy is working and turn in the pictures.
5) How many additional logic elements are used to map the policy for the SPI IP?

Part III Find out the vulnerability

6) What are the vulnerabilities of these two policies?
7) How an attacker can exploit the vulnerability to maliciously write on RAM by corrupting the lock register?
8) Implement a new policy with correct specifications and show that the vulnerability of the policies is mitigated.

Optional Follow-up

Part IV Additional Study on Attacks and Policy-based Mitigation Strategies

- Try to come up with other attack instances where the SPI IP will try to
  1. Maliciously access
  2. Snoop i.e. indirectly observe the assets
  3. Illegally modify the assets

in the given SoC model.

Also, write policies in the wrapper for each case scenarios to prevent such attacks. Implement the attacks on the design and show that the policies are working (via in-system memory content editor).

☞ Lab Report Guidelines and Demonstration

Deliverables:

1. In your report, give answers to ALL the questions.
2. Give all the code that is required.
3. Attach screenshots as required.

Demonstration:

1. For part I, show your result to TA.
2. For part II, show the results of SPI access control policy to TA.
3. For part III, show the newly written policies and results to TA.

References and Further Reading

[1] A. Basak, S. Bhunia and S. Ray, "A flexible architecture for systematic implementation of SoC security policies," 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, 2015, pp. 536-543.

[2] A. Basak, S. Bhunia and S. Ray, "Exploiting design-for-debug for flexible SoC security architecture," 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, 2016, pp. 1-6.